M.S. THESIS

Designing FPGA-based modular architectures for NLP models

자연어 처리 모델을 위한 FPGA 기반 모듈러 아키텍처 설계

BY

Hur Suyeon

FEBRUARY 2022

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

M.S. THESIS

Designing FPGA-based modular architectures for NLP models

자연어 처리 모델을 위한 FPGA 기반 모듈러 아키텍처 설계

BY

Hur Suyeon

FEBRUARY 2022

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Designing FPGA-based modular architectures for NLP models

자연어 처리 모델을 위한 FPGA 기반 모듈러 아키텍처 설계

지도교수 김 장 우

이 논문을 공학석사 학위논문으로 제출함

2021년 11월

서울대학교 대학원

전기 컴퓨터 공학부

허 수 연

허수연의 공학석사 학위 논문을 인준함

2021년 12월

위 원 장: 이 병 영

부위원장: 김 장 우

위 원: 심 재 웅

Abstract

Neural networks based natural language processing (NLP) models (e.g., LSTM, BERT) are emerging as promising solutions for NLP tasks. When running NLP models, we should support fast inference in a single batch environment, as NLP tasks require immediate responses. However, it is difficult to accelerate NLP models in a single batch due to the three challenges that follow; (1) a wide range of dimensions and irregular matrix operations, (2) non-negligible vector operations' latency, and (3) heterogeneity of vector operations.

In this paper, we propose FlexRun, an FPGA-based modular architecture approach to solve three challenges. FlexRun reconfigures the architecture adaptively to the input models. To this end, FlexRun consists of three parts. First, FlexRun:Architecture is a base architecture template with reconfigurable parameters. Next, in FlexRun:Algorithm, we define the design space and suggest algorithms to find the best design points in the design space. Lastly, FlexRun:Automation automatically finds the best design and implements the resulting architecture. For evaluation, we use Intel's high-end FPGAs and achieve 2.69× and 1.44× speedup compared to V100 and Brainwave-like FPGA baseline, respectively.

keywords: Natural Language Processing (NLP), RNN, LSTM, Transformer, BERT, Machine Learning, FPGA, Modular Architecture, Accelerator, Design space exploration

student number: 2019-24165

Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

With the emergence of Deep neural networks (DNNs), various DNN-based natural language processing (NLP) models have been developed rapidly. There are two types of DNN-based NLP models and both types show high accuracy in various NLP tasks. Recurrent neural networks [26]-based models are good at speech recognition [14] and translation [32]. Meanwhile, attention-based models are mainly used for question answering tasks.

When running NLP models, fast inference support in a single batch environment is essential, as most NLP tasks require real-time interactive services. For a quick response, we should process input as soon as it enters, in a single batch [10]. However, it is difficult to accelerate NLP models in a single batch due to the characteristics that cause challenges. Such characteristics are as follows: (1) Diverse operational complexities, (2) Varying ranges of dimensions, (3) Various parameter configurations, and (4) Heterogeneous vector operations.

The characteristics of NLP models incur three challenges (i.e., a wide range of dimensions and irregular matrix operations, non-negligible vector operations' latency, heterogeneity of vector operations). First, due to characteristics (2) and (3), we should cover a very wide range of dimensions and deal with irregular matrix operations. Next, we need to reduce the overhead of vector operations as well as matrix operations.

According to characteristic (1), attetion-based NLP models have complex vector operations, unlike RNN-based models, resulting in non-negligible overhead. Lastly, we should handle the heterogeneity of vector operations (4) as models consist of vector operations of different types, orders, and lengths.

However, existing works [12], [11], [10], [22] that accelerate NLP models cannot solve three challenges. For example, GPUs show low utilization when running NLP models in a single batch or running small models because they are throughput-oriented. Also, some works design ASICs for target models. But ASICs made for a particular model and configuration [12] perform poorly on different models or the same models with different configurations. Therefore, previous approaches degrade the performance of some NLP models or fail to support some models.

In this paper, we propose FlexRun, an FPGA-based modular architecture approach to solve three challenges of NLP models. FlexRun exploits the high reconfigurability of FPGAs to dynamically adapt the architecture to the target model and its configuration. FlexRun includes three main schemes, FlexRun:Architecture, FlexRun:Algorithm, and FlexRun:Automation.

First, FlexRun:Architecture is an FPGA-based flexible base architecture template. Our base architecture template alleviates the overhead of vector operations by adopting deeply pipelined architecture. Most importantly, it consists of parameterized predefined basic modules so that we can configure architecture to fit the input model and configuration.

Next, we carefully define the design space of the base architecture template (i.e, matrix unit: three dimensions, vector unit: vector operators' types, order, and number) considering the aforementioned three challenges. Then, we suggest FlexRun:Algorithm, design space exploration algorithms to find the best modules and parameters set for the input models.

Lastly, we propose an automatic tool, FlexRun:Automation which automates the entire steps; finding the best architecture design and implementing the architecture.

FlexRun:Automation reconfigures compute units, memory units and interconnects according to the results of FlexRun:Algorithm. Also, it generates a new decoder so that instructions can be properly decoded to a modified architecture.

For evaluation, we choose STRATIX GX and STRATIX MX as HW platforms for BERT and GPT2, respectively. As the baseline, we use GPU and Intel's Brainwave-like architecture [22] on FPGA. For GPU, we use V100 with Tensorcore enabled. Compared to the FPGA baseline, FlexRun achieves an average speedup of 1.59× on the various configurations of BERT. For GPT2, FlexRun gets 1.31× average speedup. In the case of GPU baseline, FlexRun improves the performance by 2.79× and 2.59 for BERT and GPT2, respectively. Finally, we evaluate the scalability of FlexRun by doubling the compute and memory resources of FPGAs. The results show that FlexRun is able to get the scalable performance improvement, showing 1.57× additional speedup compared to MX and GX.

Chapter 2

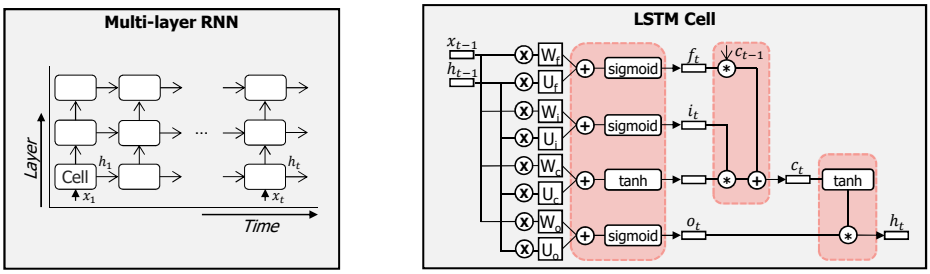
Background

2.1 Neural Networks-based NLP models

There are two types of Neural Networks-based NLP models. The first is Recurrent Neural Networks [26] (RNN)-based NLP models like SRNN, long short term memory (i.e., LSTM) [15], and gated recurrent neural networks (i.e., GRU) [26]. The other is attention-based NLP models which include Transformer [29], BERT [9], and GPT2 [23]. We use both types for various NLP tasks such as language modeling, question answering, or translation. For example, RNN-based models are mainly for speech recognition [14] and translation [32]. Meanwhile, attention-based models exhibit high accuracy in question answering tasks like SQuAD [24].

2.1.1 RNN-based NLP Models

First, we introduce RNN-based NLP models (e.g., LSTM, GRU). In Fig. 2.1a, RNNs are a structure in which cells of the same operations are repeated with incoming inputs over time. The operations in the cell differ depending on the types of RNNs, but they usually consist of matrix-vector operations and some vector operations. The cells are stacked for higher accuracy, which is called a multi-layer model. As an example, we describe the cell operations of LSTM [15]. An LSTM's cell consists of four gates (i.e.,



(a) The architecture of multi-layer RNNs.

(b) Operations of LSTM's cell.

Figure 2.1: Architecture of RNN-based models (left) and detailed cell operations of an LSTM (right).

forget, input, cell, output) and each gate has two weight matrices and one bias vector $(W_f, U_f, b_f, W_i, U_i, b_i, W_c, U_c, b_c, W_o, U_o, b_o)$ of the same dimensions ($W \in \mathbb{R}^{d_{hidden} \times d_{hidden}}$, $b \in \mathbb{R}^{1 \times d_{hidden}}$). Fig. 2.1b illustrates each gate's operations. In Fig. 2.1b, x_t, h_t, i_t, f_t, c_t, and o_t indiacte the input, state vector, input gate, forget gate, cell state, and output gate of time t, respectively. They are vectors of the same dimension, $\mathbb{R}^{1 \times d_{hidden}}$. Also, "$*, +, sigmoid, tanh\ (hyperbolic\ tangent)$" are elemente-wise vector operations while "$\times$" is a matrix-vector multlication. We highlight the vector operations with red boxes. Note that we omit bias add operations in Fig. 2.1b for simplicity.

2.1.2 Attention-based NLP Models

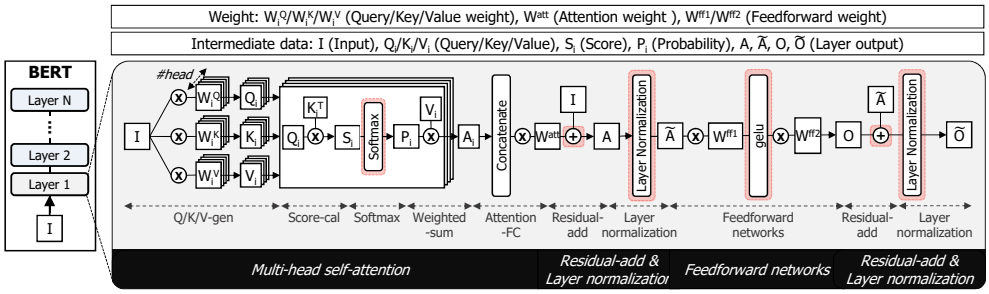


Figure 2.2: Operations for a one layer of BERT.

Next, attention-based NLP models like BERT [9] and GPT2 [23] achieve noticeably high accuracy thanks to attention operations [5] that can catch the relationship between the words. In addition to the attention operations, they include many other complex operations for higher performance.

We describe BERT as a representative example of the attention-based NLP models. As shown in Fig. 2.2, BERT consists of multiple layers and each layer has weights of diverse dimensions (e.g., $W_i^Q/W_i^K/W_i^V \in \mathbb{R}^{d_{hidden} \times d_{head}}$, $b_i^Q/b_i^K/b_i^V \in \mathbb{R}^{1 \times d_{head}}$, $W^{atten} \in \mathbb{R}^{d_{hidden} \times d_{hidden}}$, $W^{ff1} \in \mathbb{R}^{d_{hidden} \times d_{ff}}$, $b^{ff1} \in \mathbb{R}^{1 \times d_{ff}}$, $W^{ff2} \in \mathbb{R}^{d_{ff} \times d_{hidden}}$, $b^{ff2}/gamma^{1,2}$ $/beta^{1,2} \in \mathbb{R}^{1 \times d_{hidden}}$ where $i = 1 \ldots \#head$). When BERT receives input I ($I \in \mathbb{R}^{S \times d_{hidden}}$), the operations of Fig. 2.2 are repeated as many as the number of layers. We highlight the vector operations (i.e., Softmax, +, Layer Normalization, gelu) with red boxes. Vector operations like Layer Normalization are complex operations that include several basic vector operations such as addition or multiplication. Note that we omit the operations using a bias in Fig. 2.2 for simplicity, too.

Other NLP models are similar to BERT, but they have some differences. First, GPT2 [23] does the same operations as BERT, but it has dependencies between incoming input vectors. The next input vector can not start processing until the operations on the previous input vector are complete. Such structures with dependencies between the inputs are called decoders. On the other hand, BERT has no dependency between its inputs. The structures like BERT are called encoders. In the case of a Transformer [29], it includes both an encoder and a decoder, and ReLU is used as an activation function instead of gelu.

2.2 Fast inference support for NLP tasks

For NLP tasks, fast inference in a single batch environment is very important [10]. This is because NLP models are mainly for real-time interactive services like speech recogntion [14], question answering [24] and translation [32]. Interactive services re-

quire immediate responses; otherwise, their QoS will be seriously compromised. For immediate responses, we cannot deploy batch processing that collects multiple user inputs and processes them at once. Therefore, the input should be processed fastly as soon as it is entered. For example, MLPerf [25] designs four diffrerent realistic ML inference scenarios. Among them, single-stream and server scenarios assume applications like online translation where responsiveness is critical. Both single-stream and server scenarios set their batch size as 1 and use latency as the key evaluation metric.

Chapter 3

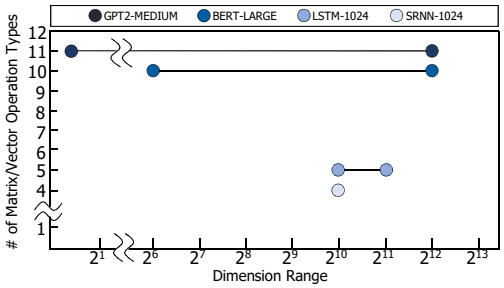
Motivation

3.1 Chracteristics of NLP models

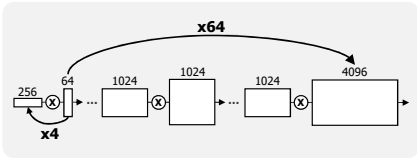
In this section, we will describe the characteristics of NLP models: (1) Diverse operational complexities, (2) Varying range of dimensions, (3) Various parameter configurations, and (4) Heterogeneous vector operations.

3.1.1 Diverse operational complexities

The operational complexities of NLP models are diverse. In Fig. 3.1a, the y-axis indicates how many different types of operations each NLP model has (e.g., gemv, gemm, transpose, exponent, sigmoid, tanh, ReLU, gelu, add/sub, multiplication, reduction, square, sqrt, reciprocal). In the figure, RNN-based NLP models include a small number of different operations, but attention-based NLP models do not. For example, BERT-LARGE has 10 different types of operations. However, LSTM-1024 or SRNN-1024 has only 5, and 4 different operations types, respectively. For details, we list the operations each NLP model contains in Table 3.1. As in Table 3.1, attention-based NLP models (i.e., BERT, GPT2) have much more complex vector operations such as exponent or reduction than RNN-based models.



(a) The number of opertaion types and range of dimensions for NLP models



(b) Some of gemm operations in BERT-LARGE and operations' dimensions.

Figure 3.1: The left figure is the number of opertaion types (y-axis) and ranges of dimensions (x-axis) for 4 NLP models, GPT2-MEDIUM, BERT-LARGE, LSTM-1024, and SRNN-1024. The word and number after the hyphen are the parameter scales of the model. For example, LSTM-1024 and SRNN-1024 indicate the LSTM and SRNN whose internal dimension size is 1024, respectively. We get the parameter size for BERT and GPT2 from the paper [9], [23]. The right figure shows some gemm (general matrix multiplications) operations in BERT and their dimensions.

3.1.2 Varying range of dimensions

NLP models have operations of varying dimensions. First, some models have to deal with a wide range of dimensions while others do not. In Fig. 3.1a, the x-axis is the dimension range of different NLP models. Dimension range is the minimum and maximum values of the operations' dimensions in the models. The figure shows that attention-based NLP models like BERT or GPT2 consist of operations of varying dimensions. For example, BERT-LARGE has dimension sizes from 64 to 4096 ($\times$64) while SRNN-1024 has only one size, 1024.

Also, gemm operations in attention-based NLP models have very irregular dimensions rather than a square. For example, Fig. 3.1b show some gemm operations in BERT-LARGE. We can observe that the first gemm operation in the figure has two dimensions, 64 and 256, that differ by four times.

Table 3.1: Matrix and vector operation types in RNN (e.g., SRNN, LSTM, GRU) and attention-based NLP models (e.g., Transformer, BERT, GPT2). *gemv* and *gemm* are general matrix-vector, and matrix-matrix multiplications, respectively. In case of the vector operations, all operations except reduction are element-wise operations. *tan* and *sig* are tangent hyperbolic and sigmoid operations, respectively. *exp*, *mul*, and *div* stand for exponential, multiplication, and division operations, respectively.

	Matrix Operations			**Vector Operations**					
	gemv	gemm	transpose	activation	exp	add/sub	mul	reduction	square/sqrt/div
SRNN	✓	-	-	tan	-	✓	✓	-	-
LSTM	✓	-	-	sig/tan	-	✓	✓	-	-
GRU	✓	-	-	sig/tan	-	✓	✓	-	-
Transformer	✓	✓	✓	ReLU	✓	✓	✓	✓	✓
BERT	-	✓	✓	gelu	✓	✓	✓	✓	✓
GPT2	✓	✓	✓	gelu	✓	✓	✓	✓	✓

3.1.3 Various parameter configurations

For each NLP model, there are diversities in parameter configurations. Table 3.2 shows some parameter configurations for BERT. In the table, there are many versions of BERT according to parameter scales, from TINY to MG3 (Megatron3 [28]). These different versions do the same operations but with totally different parameter scales. For example, BERT-MG3 uses 24× larger dimensions than BERT-TINY. Other NLP models also have various parameter configurations.

3.1.4 Heterogeneous vector operations

Each NLP model has heterogeneous vector operations. Fig. 3.2 visualizes the lists of vector operations that are executed sequentially (i.e., sequential-vector-operation, it is marked as *s* in the figure) in SRNN, LSTM, GRU, and BERT. From the figure, we can observe that most NLP models have quite heterogeneous sequential-vector-

Table 3.2: Various versions of BERT according to the parameter scales.

	Configuration	d_{hidden}	d_{head}	$\#head$	d_{ff}
BERT	TINY [27]	128	64	2	512
	MEDIUM [27]	512	64	8	2048
	BASE [9]	768	64	12	3072
	LARGE [9]	1024	64	16	4096
	MG1 [28]	1280	80	16	5120
	MG3 [28]	3072	128	24	3072

operations. For example, *s2* and *s4* of BERT greatly differ in both lengths and types of vector operations.

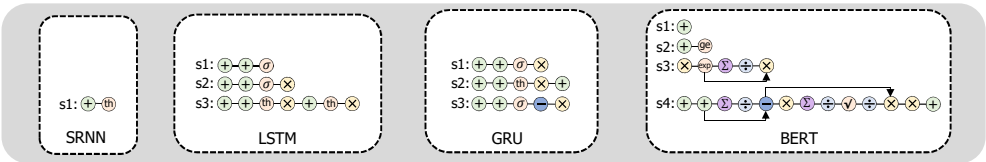


Figure 3.2: The lists of seqeuntial-vector-opearations in SRNN, LSTM, GRU, and BERT. Seqeuntial-vector-opearations are marked with s# in the figure.

We summarize the characteristics of NLP models as follows.

- Each NLP model has different computational complexities. Specifically, attention-based NLP models have much more complex and diverse operations than RNN-based NLP models.
- Attention-based NLP models have wide range of dimensions as well as highly irregular gemm operations.
- NLP models have a large diversity in terms of parameter configurations.
- The sequential-vector-operations of each NLP model are heterogeneous.

3.2 Challenges of NLP models

From the characteristics of NLP models, we derive challenges that make fast inference support difficult for diverse NLP models in a single batch environment.

3.2.1 Challenge 1: Wide range of dimensions and irregular matrix operations

First of all, some NLP models (e.g., BERT, GPT2) cover a very wide range of dimensions as shown in Fig. 3.1a. Also, in attention-based NLP models, most gemv/gemm operations are irregular (Fig. 3.1b). In addition, there are diverse parameter configurations within the same model (Table 3.2). Therefore, the NLP accelerator should run both small and big models fastly while dealing with various dimensions of irregular matrix operations. However, the conventional accelerators (e.g., big square systolic array) exhibit slow inference due to the low utilization of matrix operations as they cannot deal with diversity and irregularity of dimensions.

3.2.2 Challenge 2: Non-negligible vector operations' latency

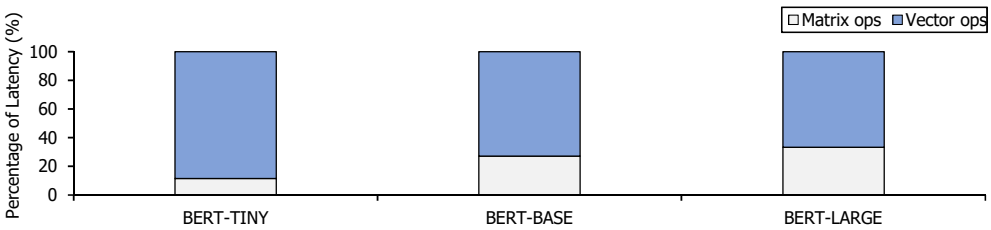


Figure 3.3: The latency breakdown of BERT-TINY/BASE/LARGE on Tesla V100. We use XLA [7], the compiler optimization provided by TensorFlow [4]. Refer to section 5.2.2 for a detailed experimental setup.

Next, complex vector operations of NLP models have non-negligible overhead. They take a dominant portion of total latency in a single batch environment. Fig 3.3

is a latency breakdown of BERT-TINY, BERT-BASE, and BERT-LARGE on GPU Tesla V100 [1]. For BERT-LARGE, vector operations (e.g., exp, gelu) take 66.7% of total latency, which is comparable to twice the gemm operations. Also, as the size of the model shrinks, the portion taken by vector operations grows. In BERT-BASE and BERT-TINY, vector operations take 72.9% and 88.5% of total latency, respectively. Note that we apply XLA [7], the compiler optimization of TensorFlow [4] that minimizes the vector operations' latency by layer fusion. Therefore, to achieve high performance for NLP models, we should further reduce the overhead of vector operations.

3.2.3 Challenge 3: Heterogeneity of vector operations

As shown in the Fig. 3.2, the NLP models consist of vector operations of different types, orders, and lengths. For example, there is no overlap between BERT's sequential-vector-operations (i.e., *s1* ~ *s4*) and GRU's sequential-vector-operations (i.e., *s1* ~ *s3*). Also, sequential-vector-operations vary within the same model too. In Fig. 3.2, BERT's *s3* and *s4* are different in types, orders, and lengths of the vector operations. So, to accelerate the NLP models, it is essential to efficiently support all models' vector operations.

3.3 Limitations of previous works

There have been many works to accelerate the NLP models. They generally take two approaches, exploiting general-purpose accelerator (i.e., GPU) or designing the specialized architecture for the specific models (i.e., ASICs). However, those approaches cannot properly deal with the three challenges that originate from the models' characteristics. Therefore, these approaches degrade the performance of some NLP models or fail to support some models.

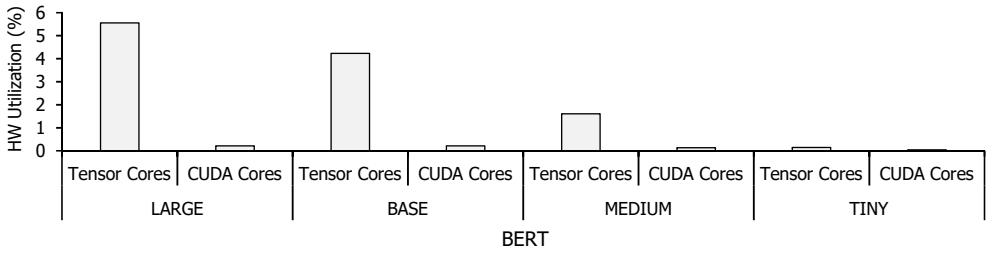


Figure 3.4: Tesla V100's (Tensor Cores and CUDA Cores) utilization on different versions of BERT (i.e., BERT-LARGE, BERT-BASE, BERT-MEDIUM, and BERT-TINY).

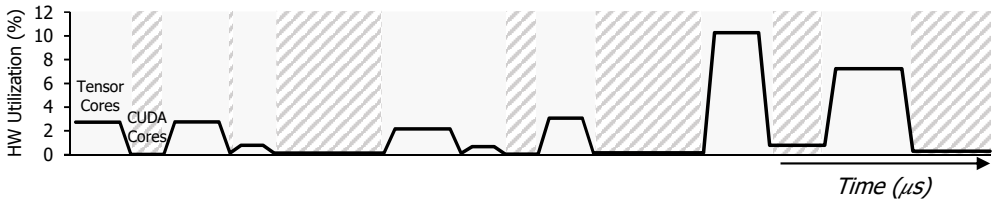


Figure 3.5: The utilization of V100 on BERT-LARGE over time. The gray boxes are gemm operations using Tensor Cores and the shaded boxes are vector operations using CUDA Cores.

3.3.1 GPU (general-purpose accelerator)

GPUs are the most commonly used accelerator for various Deep Neural Network (DNN) models. Thanks to a convenient framework and highly parallel architecture, GPUs can support various models and achieve high performance for big models or models with large batch size [20]. However, GPUs cannot well handle the NLP models in the single batch environment. Fig. 3.4 shows the average utilization of Tensor Cores (gemv/gemm compute units) and CUDA Cores (vector compute units) in V100 for different versions of BERT, assuming a single batch environment. We observe that GPUs are severely underutilized in a single batch environment even for a large model (Tensor Cores: 5.55%, CUDA Cores: 0.21% for BERT-LARGE). Also, utilization dramatically decreases as the model's size diminishes. For example, BERT-BASE has

4.23% and 0.22% effective utilization for Tensor Cores and CUDA Cores, respectively while BERT-TINY has only 0.15% and 0.05%.

In addition, GPUs cannot deal with vector operations efficiently as they mainly focus on gemm operations. Fig. 3.5 is the change in utilizations over time when running single batch BERT-LARGE on V100. In Fig. 3.5, the latency of vector operations (CUDA Cores) is exposed and takes a comparable portion to twice the gemm operations (Tensor Cores). Also, utilization of vector operations is very low (0.78% at maximum). This severe underutilization is due to the working mechanism of GPU, where data go down into memory between every gemm and vector operation, resulting in frequent memory access. In conclusion, GPUs are not adequate for running NLP models in a single batch environment.

3.3.2 ASICs

Previous works [12], [11] make ASICs for specific models. ASICs usually show a great performance for a target model and configuration. They are exploiting specific characteristics of the target model and have the best setting for the target configurations. However, ASICs fail to resolve challenge 1 (Diversity and irregularity of dimensions) and challenge 3 (Heterogeneous vector operations) of NLP models as they are fixed architectures.

First, ASICs optimized for a specific model and configuration may not show good performance for the same models with different parameter configurations. However, there are diverse parameter configurations for the same NLP models. Also, we can arbitrarily change parameters as in previous studies [28], [27]. For example, A^3 [12] focuses on the attention operations of NLP models, making attention-specialized units. A^3 sets one of its specialized units' size as the same as d_{head} size (64) of BERT-LARGE/BASE, which are its target models. However, if we change d_{head} size to 32 [28], utilization will cut in half.

Also, some ASICs may fail to run NLP models that they do not cover. For example,

ASICs optimized for LSTM cannot run the attetion-based NLP models like BERT due to the absence of required units (e.g., transpose, reduction, sqrt). They can run the models by adding a few units, but it will be inefficient without solutions to deal with the heterogeneity. Also, it is impossible to add new units for every upcoming model, as new models are being released at a rapid pace.

3.3.3 FPGA

There have been some previous works that exploit FPGAs for accelerating NLP workloads [10], [22]. As advantages in running DNN workloads, FPGAs have high reconfigurability. Also, FPGAs support various data precision (e.g., FP32, INT8) and some products have HBM on chip [6]. New FPGA products with many operators and large on-chip memory for DNN workloads are actively entering the market recently [6]. However, the previous works using FPGA have also focused on accelerating specific NLP models (e.g., LSTM) and fail to solve challenges of NLP models.

3.4 Solutions

We summarize the previous works and their weaknesses in Table 3.3. As we have already mentioned, some previous works [12] cannot efficiently handle the diversity in dimensions (challenge 1) neither do they deal with overhead and heterogeneity of vector operations (challenge 2, challenge 3) in NLP models. Meanwhile, other works [10], [22], [11] relieve the overhead of vector operations by pipelining, solving challenge 2. However, they cannot still cope with challenge 1 and challenge 3.

To solve the challenges, we propose a modular architecture approach. In the modular architecture approach, the architecture composes of pre-defined basic modules so that we can flexibly reconfigure the architecture adaptively to the target model. To this end, we make FlexRun, the end-to-end solution that implements efficient modular ar-

Table 3.3: Previous works and their limitation in solving the challenges of NLP models.

	Previous Works	**Challenge 1**	**Challenge 2**	**Challenge 3**
Acclerator for NLP	Brainwave [10] / NPU [22]	X	O	X
	A^3 [12]	X	△	X
	ATT [11]	△	O	X

chitecture, solving the three challenges. Also, we choose FPGAs as our HW platform due to their high reconfigurability. FlexRun includes three main features. First feature is flexible base architecture template that consists of pre-defined parameterized modules (i.e., FlexRun:Architecture). The next feature is a carefully defined design space and design space exploration algorithm to find the best modules and parameter set for the target models (i.e., FlexRun:Algorithm). Finally, the last feature is an automatic tool that automates the two steps, finding the best architecture according to the inputs and implementing architecture (i.e., FlexRun:Automation). We will explain our FlexRun in detail in the Section 4. For FlexRun to achieve high performance for NLP models, we set the design goals as follows.

Design Goals

- Devise flexible architecture template that can solve three challenges (FlexRun:Architecture).
- Define design space for handling challenges 1, 3 and make algorithm to efficiently search the design space for the given inputs (FlexRun:Algorithm).
- Automate the whole steps, from searching design space to implementing architecture, for ease of use (FlexRun:Automation).

Chapter 4

FlexRun

4.1 Overview

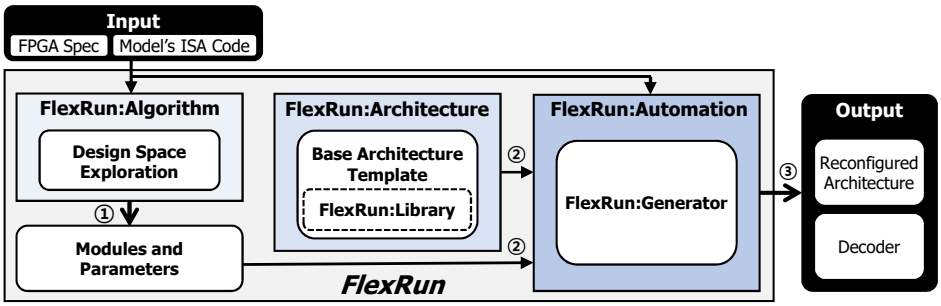
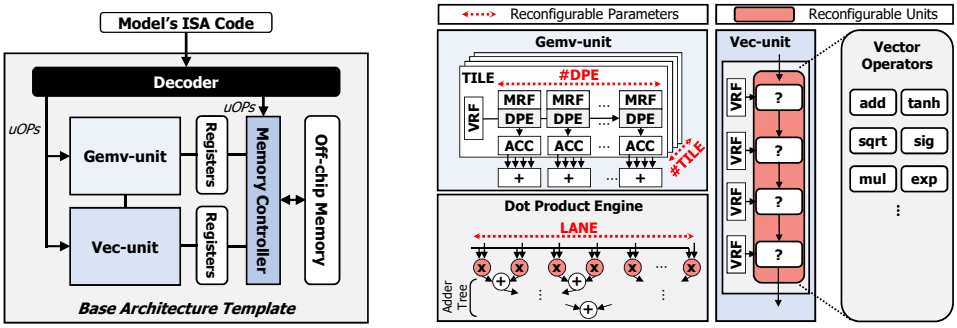


Figure 4.1: End-to-end workflow of FlexRun.

In this subsection, we will briefly explain how FlexRun works. FlexRun is an end-to-end solution that implements efficient modular architecture for NLP models. Fig. 4.1 shows the end-to-end workflow of FlexRun. FlexRun has three main schemes, FlexRun:Architecture, FlexRun:Algorithm, and FlexRun:Automation. First, FlexRun:Architecture is a base architecture template consists of pre-defined parameterized basic modules (FlexRun:Library). Next, FlexRun:Algorithm finds the best set of modules and their parameters for the given NLP model and FPGA spec (① in Fig. 4.1). Finally, a FlexRun:Automation reconfigures and implements the architecture template given by FlexRun:Architecture

according to the FlexRun:Algorithm's results (② & ③ in Fig. 4.1). Also, the FlexRun:Automation generates the new decoder codes so that the model's code is decoded for the reconfigured architecture.

4.2 FlexRun:Architecture



(a) FlexRun's base architecture template.

(b) Compute units of FlexRun's base architecture template.

Figure 4.2: FlexRun's base architecture template and details of copmpute units (Gemv-unit and Vec-unit).

FlexRun:Architecture is our base architecture template made of pre-define parameterized basic modules (FlexRun:Library). We devise a FlexRun:Architecture, considering the three challenges of NLP models.

In making our base architecture template, we refer to NPU [22]. Fig. 4.2a illustrates FlexRun's base architecture template. The base architecture template consists of three parts. First, there are two compute units: Gemv-unit and Vec-unit. Gemv-unit is a highly parallel compute unit for gemv operations. Vec-unit is a compute unit for vector operations, which can be made of any combination of basic vector operators (e.g., mul, add, exp), realizing flexibility. Note that Gemv-unit and Vec-unit have their own registers to store the weights and intermediate data. Second, there is a memory controller that prefetches required weights from off-chip memory to registers according to de-

coded uOps. Lastly, there is a decoder that decodes the model's ISA code to uOps for the memory controller and compute units. The detailed structure and working mechanism of the base architecture template are explained in the following subsections.

4.2.1 Structure of of FlexRun:Architecture

Fig. 4.2b is a detailed figure for the compute units of FlexRun's base architecture template.

Gemv-unit: Gemv-unit composes of multiple SIMD arithmetic units for vector-matrix multiplications as in Fig. 4.2b. Gemv-unit is a highly parallel architecture so that it fits for NLP workloads with many matrix operations. Also, Gemv-unit computes in vector-matrix granularity, which is good for NLP workloads that have dependencies between the inputs (decoder structures like LSTM and GPT2) in a single batch environment.

In Fig. 4.2b, Gemv-unit has multiple TILEs that split a matrix into sub-column blocks, and each TILE has several DPEs (Dot Product Engines) and ACCs (Accumulators). Each DPE executes the same number of element-wise multiplications as the size of LANEs. In our architecture template, the number of TILE, DPE, and size of LANE is reconfigurable. Therefore, through design space exploration, we can reconfigure these parameters adaptively for the target model's operations sizes, solving challenge 1 (i.e., wide ranges of dimensions and irregular matrix operations).

Also, Gemv-unit and Vec-unit are deeply pipelined, relieving the overheads of vector operations. In addition, all vector operators in Vec-unit are pipelined so that there is no unnecessary memory access between vector operations (Fig. 4.3b). Therefore, most vector operations' latency is hidden by gemv operations' or other vector operations' latency, mitigating challenge 2 (i.e., vector operations' overhead).

Vec-unit: Vec-unit executes vector operations of size VEC_LANE at once. In our template, we set the size of VEC_LANE equal to the Gemv-unit's LANE size. Also, there are some additional operators (i.e., reduction, exp, gelu) compared to [22] to

support attention-based NLP models. Gemv-unit and Vec-unit have direct datapaths for vector pipelining. As a result, the Vec-unit can start independent instruction upon receiving the first sub-block results from the Gemv-unit.

Previous works [22] have fixed types, numbers, and order of vector operators in its vector compute units. However, as shown in Fig. 4.2b, we remain Vec-unit as an empty box which can be made of any combinations of basic vector operators in FlexRun:Library. In this way, we can efficiently deal with challenge 3 (i.e., heterogeneity of vector operations). Some may think that the flexible vector compute units are unnecessary as pipelining alleviates the overhead of vector operations. However, if the vector compute units are not properly configured, the overhead of vector operations is exposed even with the pipelining. Details are provided in the section 4.3 and 4.4.

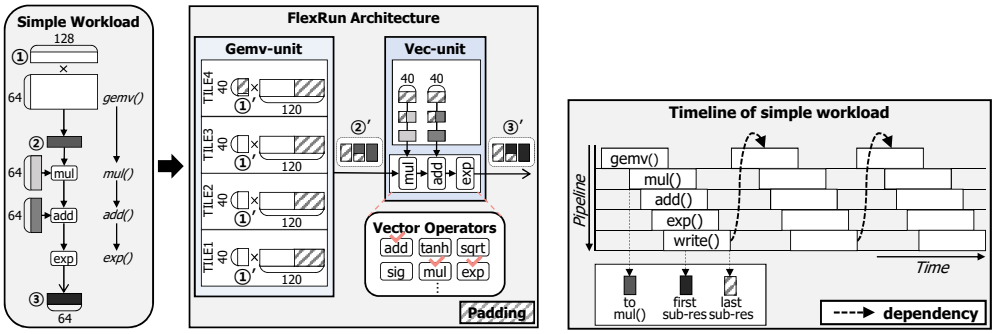
Memory and Datapath: Gemv-unit and Vec-unit have separate register files (MRF and VRF, respectively) for decoupled execution. Previous works [10], [22] which utilize the persistent-AI keep all weights in on-chip memory (MRFs). However, some NLP models (e.g., BERT-LARGE) cannot hold their whole weights in on-chip memory due to their excessive size. Therefore, we add new datapaths which connect memory controller and MRFs to fetch weights from off-chip memory to MRFs. These datapaths are also used when we use the results of Gemv-unit or Vec-unit as the vector input of Gemv-unit. Also, we place a matrix transpose unit in the MRF/VRF write-back path.

ISA and Decoder: FlexRun's ISA extends the NPU's ISA [22] to support new operations of attention-based NLP models (i.e. reduction, exp, gelu, transpose). FlexRun's ISA is architecture-independent for programmability. Instead, the decoder decodes the ISA into multiple uOps adaptively to the reconfigured architecture. When we reconfigure the architecture, we remake the decoder to fit the new architecture. We will provide ISA and decoder examples in section 4.5.

FlexRun:Library In FlexRun:Library, there are basic modules of our template; TILE of Gemv-unit and vector operators for Vec-unit. These basic modules are parameterized and modular so that we can configure and merge them adaptively to the

target model.

4.2.2 Working mechanism of FlexRun:Architecture



(a) Simple workload and execution of workload in FlexRun.

(b) Timeline of simple workload's execution.

Figure 4.3: Simple workload and FlexRun's executions of the workload. The left figure is the timeline graph of the workload when it is repeated three times with dependencies.

For smooth understanding, we assume a simple workload as Fig. 4.3a. The workload consists of one gemv operation and following vector opeartions; gemv()-mul()-add()-exp(). The size of gemv operation is $(1\times128)\times(128\times64)$. For the example workload, we configure the FlexRun's architecture as follow: In the case of Gemv-unit, we assume (#TILE, #DPE, LANE size) as (4, 120, 40). Also, we set the Vec-unit's structure as [mul-add-exp], which is the best design for the example.

Now we explain how our architecture handles the example workload. First, in Fig. 4.3a, we add some paddings to the input vector (①) so that it would be the multiple of LANE size (128→160). Then, the input vector is distributed to each TILE (①→①'). All TILEs have the same size of vectors, 40. But, 32 elements of TILE4's input vector are zero due to padding. Also, we add paddings to the row of weight matrix's to be the multiple of #DPE (64→120). Next, the DPEs and ACCs perform vector-matrix multiplications of LANE size at once. Since the size of Gemv-unit's ouput vector (②) is 120, Gemv-unit produces three (120/40) sub-vectors of LANE

size (②→②'). These sub-vectors (②') are fed into the Vec-unit as soon as they come out from Gemv-unit. Finally, the result vector (③, ③') comes out from the Vec-unit. As shown in the figure, a lot of fragmentation occurs as the size of matrix and vector does not match the (#TILE, #DPE, LANE size).

Fig. 4.3b shows the timeline and pipeline graph of the simple workload when it is repeated three times with dependencies. Gemv-unit and Vec-unit are pipelined so that the output sub-vectors of Gemv-unit directly go to the Vec-unit even though Gemv-unit does not complete its executions. These pipelined execution helps hide the latency of vector operations with gemv operations. Each vector operator is also deeply pipelined, hiding each other's latency. However, due to the dependencies, the next input's execution cannot start until the previous one is finished. If there is no dependency, the next input can start processing as soon as the compute unit becomes free.

4.3 FlexRun:Algorithm - Design Space

Table 4.1: Design space of FlexRun.

Units	Design Space	Explanation
Gemv-unit	#TILE	The number of TILE
	#DPE	The number of DPE
	LANE size	The size of LANE
Vec-unit	Type	Vector operators to choose
	Order	Vector operators placement order
	Number	The number of same vector operators to select

Table 4.1 shows FlexRun's design space. FlexRun aims to find the best combination in the design space for the given model and FPGA spec. In case of the Gemv-unit, three parameters (#TILE, #DPE, LANE size) are in our design space. These parameters cover all the gemv dimensions, $(1\times N)\times(N\times K)$. For the Vec-unit, the types, number,

and order of the basic vector operators are in design space. In the following contents, we will show how our design space choice affects the performance of the NLP models and solves the three challenges.

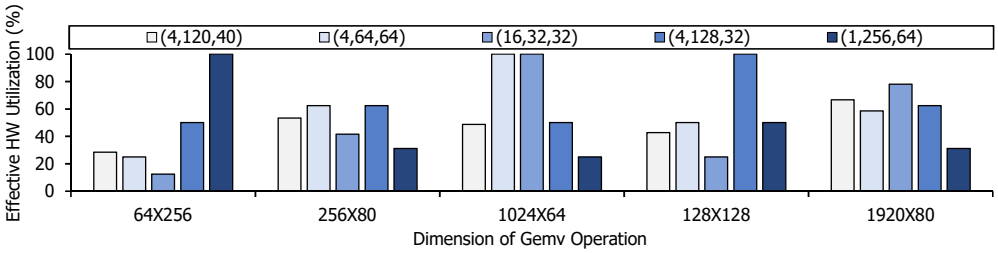


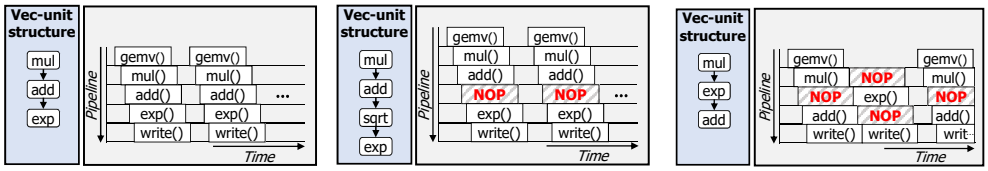
Figure 4.4: The effective HW utilization of Gemv-unit for various dimensions of gemv operations. The legend is the reconfigurable parameters of Gemv-unit: (#TILE, #DPE, LANE size). To be specific, (64×256) in the x-axis indicates the gemv operation that the vector size is (1×64) and the matrix size is (64×256).

4.3.1 Design space of Gemv-unit: (#TILE, #DPE, LANE size)

Fig. 4.4 shows the HW utilization for gemv operations with various dimensions, changing the three Gemv-unit's parameters. The x-axis is the dimension of gemv operation and the y-axis is the effective utilization of Gemv-unit. Note that these gemv operations are from the real NLP workloads. Also, the legend is the combination of (#TILE, #DPE, LANE size). All combinations satisfy the same maximum resource limitation. In the figure, each gemv operation has different utilization according to the combinations of (#TILE, #DPE, LANE size). For example, in the case of (64×256), it achieves 100% effective utilization when the (#TILE, #DPE, LANE size) is (1, 256, 64), while (16, 32, 32) reduces the utilization by 87.5%. Also, there is no combination of (#TILE, #DPE, LANE size) which works best for all gemv operations. For example, (1, 256, 64) works best for (64×256), but has the worst utilization for (1024×64). From the experiments, we conclude that fixed-size matrix compute units cannot handle a wide range of dimensions as well as irregular matrix operations. Therefore, by configuring

(#TILE, #DPE, LANE size) adaptively to the model, we can get the highest possible utilization, resolving challenge 1.

4.3.2 Design space of Vec-unit: types, number, and order of basic vector operators

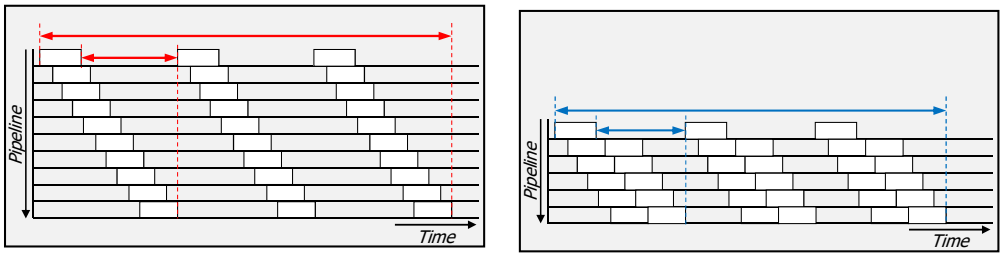


(a) Right types & order. (b) Wrong types but right order. (c) Right types but wrong order.

Figure 4.5: Timeline and pipleine graphs of simple workload's executions according to Vec-unit's structures.

When we design vector compute units, we need to consider the types, order, and the number of vector operators. We will explain how Vec-unit's design affects the performance of NLP models by revisiting the simple workload, gemv()-mul()-add()-exp() in section 4.2.2. The simple workload needs three types of operators: mul, add, and exp. Fig. 4.5 shows the timeline and pipeline graphs of simple workload executions for the three cases; right types & order (Fig. 4.5a), wrong types but right order (Fig. 4.5b), and right types but wrong order (Fig. 4.5c). First, Fig. 4.5a is when the types and order are both perfectly matched to the given model. However, if there are any unnecessary operators (wrong types), it occurs underutilization like Fig. 4.5b. Also, in the case when the order is not optimal, the data may go through the pipeline again to complete the operations, as in Fig. 4.5c.

Depending on the number of vector operators which determines the pipeline's depth, a trade-off occurs like Fig. 4.6. In the case where Vec-unit has many vector operators, the input data can be processed in one execution, as shown in Fig. 4.6a. However, in Fig. 4.6a, the pipeline gets deeper and the overhead of the exposed pipeline's depth



(a) Many operators (pipeline's depth ↑).

(b) Fewer operators (pipeline's depth ↓).

Figure 4.6: Trade-off according to the number of vector operators.

grows. Otherwise, if Vec-unit has fewer vector operators like Fig. 4.6b, the data has to go through the pipeline again to finish execution. But pipeline gets shorter and the overhead of the exposed pipeline's depth is reduced.

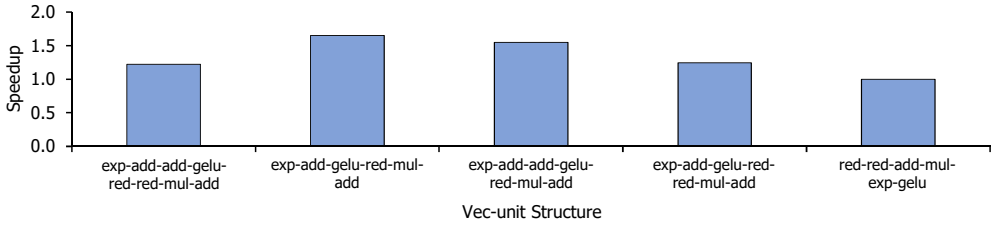


Figure 4.7: The GPT2-MEDIUM's relative performace improvement according to various Vec-unit's structures. The performance improvement is normalized to the last feature: [red-red-add-mul-exp-gelu].

We check the trends mentioned above by running GPT2. Fig. 4.7 shows the performance of GPT2-MEDIUM on various Vec-unit's structures. The x-axis is the Vec-unit's structure and the y-axis is speedup normalized to the last structure. The last structure in Fig. 4.7, the one with the random ordering has the worst performance. The first and third structures have proper ordering, but they show lower performance than the second one as they have too many operators, exposing more pipeline depth. The second structure which has optimal order and number of operators has the best performance. From the result, we conclude that by reconfiguring the Vec-unit adap-

tively to the given model, we can deal with challenge 3 (i.e., heterogeneity of vector operations).

Some may wonder it is more convenient to put the all required vector operators for the NLP models into Vec-unit and make all-to-all connections. However, in that case, there are two main problems. First, interconnect overheads will be too expensive because complex models like BERT needs 8 different operators. Additionally, future NLP models may require new types of operators, which further increases the interconnection overhead. Second, simple models like SRNNs are suffered from underutilization because they do not use most vector operators.

4.4 FlexRun:Algorithm - Design space exploration

To find the best design for the model, we explore all possible design points. To this end, we introduce two algorithms for efficient design space exploration: Gemv-unit Rearrangement for Gemv-unit and Vec-unit Reconstruction for Vec-unit. For design space exploration, we use an analytical model, a time-accurate simulator that gets the template's parameters and model's code as inputs and measures the execution latency.

4.4.1 Gemv-unit Rearrangement

Gemv-unit Rearrangement gets the model's ISA code and FPGA spec (i.e., BRAM, DSP, LUT, Memory BW) as inputs. Then, it searches all possible combinations of (#TILE, #DPE, LANE size) which satisfy the resource limitation. Among the found combinations, the algorithm selects the top-k sets which have the smallest total gemv/gemm latency of the model using the analytical models. In experiments, results of Gemv-unit Rearrangement for FPGAs on the current market are available in seconds.

Algorithm 1: Vec-unit Reconstruction Algorithm.

```
input  : model_code, (#TILE, #DPE, LANE size).
/* Get vector-opearation-sequences in the model.                    */
{vec_seq_0, ..., vec_seq_k} = get_vector_op_sequences(model_code)
/* Find Shortest Common Sequence (SCS) for given vector sequnces,
   {vec_seq_0, vec_seq_1, ..., vec_seq_k}.                          */
{SCS_0, ..., SCS_m} = Find_SCS(vec_seq_0, vec_seq_1, ..., vec_seq_k)
min_lat = total_lat(SCS_0, model_code, (#TILE, #DPE, LANE size))
optimal_structure = SCS_0
for SCS in {SCS_1, ..., SCS_m} do
    /* Get all possible subsequnces of SCS by removing duplicate
       operators.                                                   */
    {SCS_subseq_0, ..., SCS_subseq_n} = Find_SCS_subsequences(SCS)
    /* Find SCS_subsequence that gives the minimum total latency for
       the inputs.                                                  */
    for arch in {SCS_subseq_0, ..., SCS_subseq_n} do
        temp = total_lat(arch, model_code, (#TILE, #DPE, LANE size))
        if temp < min_lat then
            min_lat = temp
            optimal_structure = arch
    end
end
return optimal_structure
```

4.4.2 Vec-unit Reconstruction

Vec-unit Reconstruction gets two inputs, the model's ISA code and results of Gemv-unit Rearrangement. For target model and configurations, Vec-unit Reconstruction finds a set of (#TILE, #DPE, LANE size) and Vec-unit's structure with minimum total latency. Vec-unit Reconstruction uses the Shortest Common Sequence (SCS) algorithm. SCS is the shortest sequence which includes all the vector-operation-sequences in the model while keeping the original order of each sequence.

Algorithm 1 shows each step of Vec-unit Reconstruction. In Algorithm 1, first we

extract the list of vector-operation-sequences (vec_seq_i) from the model's code: line1. Then, we find all SCSs for given vector-operation-sequences: line2. Next, for every SCS in the list, we repeat lines 6-12. In line 6, we derive all possible sub-sequences of the SCS (SCS_subseq_i), which contain all the required operators for the model. Lastly, we find the sub-sequence which gives the minimum total execution latency for the model ($optimal\_structure$), using an analytical model, $total\_lat()$: line 7-11. For easy understanding, we visualize each step of the algorithm in Fig. 4.8.

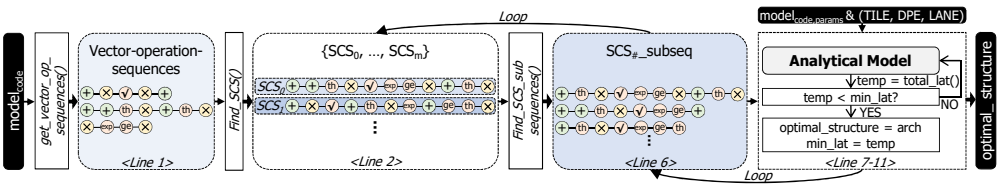


Figure 4.8: Visualization for each step of Vec-unit Reconstruction algorithm.

Results of design space exploration

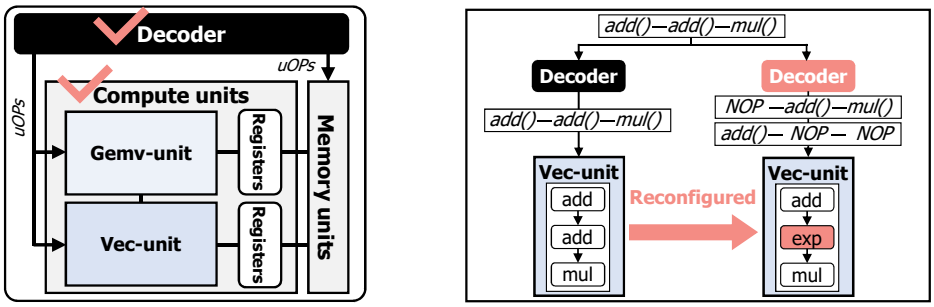
Table 4.2 shows the results of design space exploration for BERT-LARGE and GPT2-MEDIUM. In Table 4.2, two models have different set of optimal (#TILE, #DPE, LANE size) and Vec-unit's structure. Also, we can observe the trends mentioned in Section 4.3.2. Because GPT2 has dependencies between the inputs while BERT does not, the optimal Vec-unit's structure for GPT2 is shorter than that for BERT-LARGE.

Table 4.2: The results of design space exploration for GPT2-MEDIUM and BERT-LARGE.

	(#TILE, #DPE, LANE size)	**Vec-unit's structure**
BERT-LARGE	(4, 64, 64)	add-act-add-red-red-mul-add
GPT2-MEDIUM	(3, 64, 64)	add-act-add-red-mul

4.5 FlexRun:Automation

FlexRun:Automation is an automatic tool of FlexRun to make the complex reconfiguration process automatic. FlexRun:Automation consists of two main features, compute units generators and decoder generators (FlexRun:Generator). As in Fig. 4.1, FlexRun:Automation receives FlexRun:Algorithm's results (i.e., (#TILE, #DPE, LANE size) and Vec-unit's structure) as inputs and produces the reconfigured architecture and decoder accordingly.



(a) Components to be reconfigured (marked) in the base architecture template.

(b) Example for how decoder decodes the ISA according to the architectures.

Figure 4.9: The necessities of FlexRun:Generator.

4.5.1 FlexRun:Generators

When we reconfigure our base architecture template for the new target model, two components marked in Fig. 4.9a should be modified. One is the whole compute processor, including compute units (Gemv-unit and Vec-unit) as well as registers and interconnects (e.g., MRFs, VRFs, interconnects between VRFs and Vec-unit). The other is the decoder which decodes ISA into multiple uOPs for the new architecture. Fig. 4.9b shows how the decoder decodes the same ISA (i.e., ISA: add()-add()-mul()) when the Vec-unit changes. In the original structure, the order and types of vector operators fit the example ISA so that decoder simply decodes ISA into three uOPS: add()-add()-

mul(). However, when the structure changes to [add-exp-mul], decoder should decode the example into six uOPs: NOP-add()-mul() and add()-NOP-NOP.

Chapter 5

Implementation

5.1 FlexRun

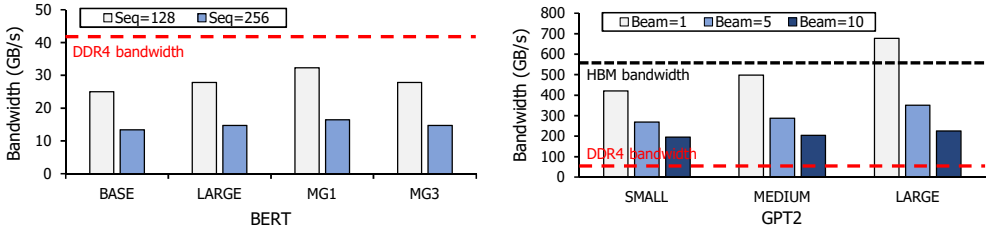
5.1.1 FlexRun

To implement basic modules in FlexRun:Library (TILE of Gemv-unit and vector operators in Vec-unit), decoder, and memory units, we use C-based Vivado High-Level Synthesis (HLS). First, TILE includes DPEs, accumulators, and registers (MRFs). We make the number of DPEs and accumulators as reconfigurable as well as LANE size. Also, we could set the registers' size adaptively to the input model and configurations. Next, we code each vector operator separately so that we can make any combinations of operators. The vector operator has a parameter named VEC_LANE (basic unit of processing), which has the same size as the Gemv-unit's LANE size.

We use python to implement two algorithms in FlexRun:Algorithm. First, Gemv-unit Rearrangement gets the model's ISA code, model descriptions (e.g., parameters, vector operations), and FPGA spec as inputs. Then it lists out all combinations of (#TILE, #DPE, LANE's size) satisfying the resource limitation. These lists go as inputs to the analytical model, and the analytical model finds the top-k combinations which have the smallest gemv operation's latency for the target model. In the evaluation, we use 3 as the value of k. Second, Vec-unit Reconstruction function receives

outputs of Gemv-unit Rearrangement as well as model's ISA code and model descriptions. Following Algorithm 1, the function finds the best set of Vec-unit's structure and (#TILE, #DPE, LANE size).

Put all together, the top generator function gets the model's ISA code and FPGA spec. Then, it gets the optimal configurations for the given model using the two functions in FlexRun:Algorithm as mentioned in the previous paragraph. Finally, two generator fucntions make the compute processor and decoder according to the given configuration.



(a) Comparison between bandwidth requirement of BERT and DDR4's bandwidth.

(b) Compariosn between bandwidth requirement of GPT2 and HBM's bandwidth.

Figure 5.1: Comparison of NLP models' bandwidth requirement and off-chip memory bandwidth of STRATIX GX (DDR4) and MX (HBM).

5.1.2 Memory

Since data reuse is limited in a single batch environment, high memory bandwidth is required to avoid the memory bottleneck. When the model's size is small enough to be stored in on-chip memory, the persistent AI approach used by previous works [10], [22] can address the memory bottleneck issues. However, this approach cannot be applied to some NLP models with large parameters (e.g., BERT-LARGE). In this work, we implemented layer-wise double buffering in on-board DDR4 and HBM to hide the memory overhead. With this technique, the memory controller prefetches the weight matrices of the next layer while the current layer is computed. Also, we set the size of registers large enough to hold these prefetched weight matrices and intermediate data.

Fig. 5.1 shows the maximum memory bandwidth requirement of our target NLP models when adopting the layer-wise double-buffering scheme. In the figure, we compare the bandwidth requirement of the models with the off-chip memory's bandwidth of two FPGAs. We use two FPGAs, Intel's STRATIX GX with DDR4 and STRATIX MX with HBM. In Fig. 5.1a, BERT with large parameters (BASE, LARGE, MG1, and MG3) can be executed on both GX and MX. However, GPT2 should be executed on an MX with HBM to avoid memory bottleneck, like Fig. 5.1b. Therefore, we use GX for BERT and MX for GPT2 in evaluation.

5.2 Workloads and Experimental Setup

Table 5.1: NLP models and their configurations included in the workloads.

	Configuration	d_{hidden}	d_{head}	*#head*	d_{ff}	*beamwidth*
LSTM	-	1024	-	-	-	-
BERT	TINY	128	64	2	512	-
	MEDIUM	512	64	8	2048	-
	BASE	768	64	12	3072	-
	LARGE	1024	64	16	4096	-
	MG1	1280	64	16	5120	-
	MG3	3072	64	24	3072	-
GPT2	TINY	768	64	12	3072	5, 10, 40
	MEDIUM	1024	64	16	4096	5, 10, 40
	LARGE	1280	80	16	5120	5, 10, 40

5.2.1 Workloads

Our target workloads and their configurations are in Table 5.1. We choose the workloads by the following criteria. First, we evaluate the attention-based NLP models

which have complex vector operations so that we can show how FlexRun reduces the overhead of vector operations (challenge 2). Next, we show how FlexRun deals with the heterogeneity of vector operations by comparing the performance trends of the three models (challenge 3). Lastly, by covering models with a wide variety of parameter scales, we show that FlexRun can cope with a wide range of dimensions and irregular matrix operations (challenge 1).

5.2.2 Experimental setup

Table 5.2: Configurations of Baseline.

FPGA	(#TILE, #DPE, LANE size)	Vec-unit's structure	Frequency
STRATIX GX	(4, 120, 40)	red-add-act-mul	275MHz
STRATIX MX	(4, 80, 40)	red-add-act-mul	290MHz

As the baseline, we use GPU and our base architecture template. For GPU, we use a Tesla V100 with Tensor Core enabled [1]. We use official open-source TensorFlow implementation of NLP models from NVIDIA and OpenAI [2]. We set the frequency of V100 as 1350MHz. When we analyze the results of GPU, we exploit the NVIDIA Nsight Systems [3] which is a performance analysis tool officially provided by NVIDIA. In the case of the second baseline, we do not apply three schemes of FlexRun. Also, we refer to NPU [22] for setting the configurations ((#TILE, #DPE, LANE size) and Vec-unit's structure). The detailed configurations are specified in Table 5.2. For the rest of the paper, we will simply call the second baseline the Baseline. For FlexRun's evaluation, we use two FPGAs, Intel's STRATIX GX and MX [8]. Also, FlexRun supports an 8-bit integer data type as NPU.

Chapter 6

Evaluation

We evaluate our schemes using a cycle-accurate simulator to measure the scalability of FlexRun, considering the trends of FPGAs with increasing resources. We carefully validate our simulator as follows. First, we implement SW-based NPU-like architecture and compares its RNN/LSTM performance against Intel's pre-validated STRATIX 10 GX and MX FPGA implementation [22]. The errors between our simulator and FPGA implementation are under 0.1% for various parameter settings. Then we add FlexRun's specific features (e.g., transpose unit) on top of SW architeture to make our FlexRun base architecture template.

6.1 Performance improvement of FlexRun compared to the Baseline

We first measure the speedup of FlexRun for BERT and GPT2, compared to the Baseline. Fig. 6.1 and Fig. 6.2 show the comparison results for BERT and GPT2, respectively. The x-axis is the parameter scale and the y-axis is the speedup. By applying the three optimization schemes of FlexRun one by one, we show how much gain each optimization brings. The legend of the figure is the optimization schemes we apply. The first legend is the Baseline. The second legend (Gemv-unit Rearrange) indicates the

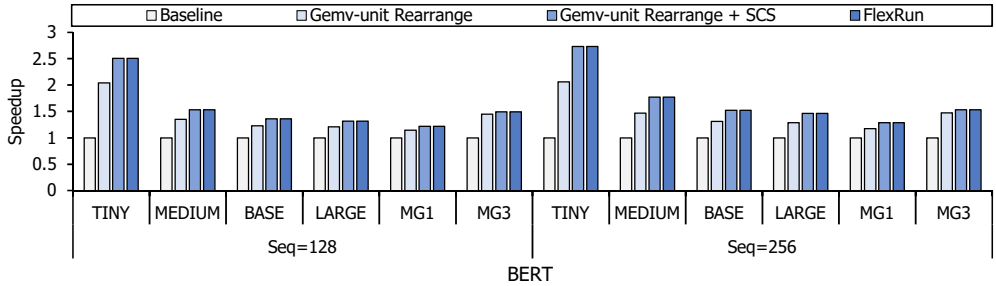


Figure 6.1: Speedup of FlexRun for BERT normalized to the Baseline.

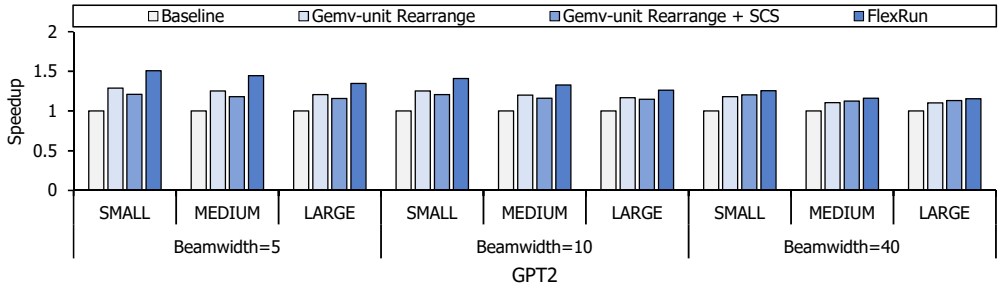
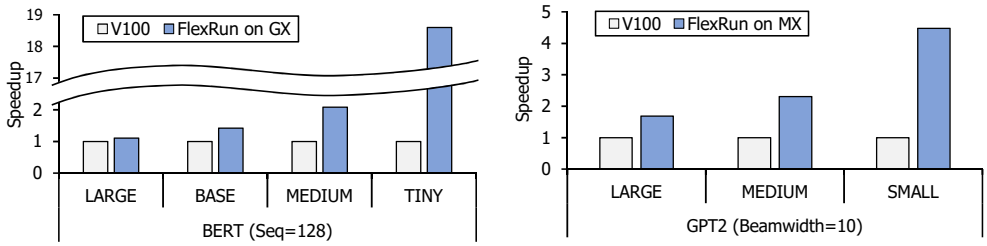


Figure 6.2: Speedup of FlexRun for GPT2 normalized to the Baseline.

case that we only apply Gemv-unit Rearrangement to the Baseline. The third legend (Gemv-unit Rearrange + SCS) is the result of changing the Vec-unit's structure to SCS in addition to the second legend. The last legend (FlexRun) is the case we apply both Gemv-unit Rearrangement and Vec-unit Reconstruction.

Fig. 6.1 is the results for BERT with two input sequence sizes: 128, and 256. First, Gemv-unit Rearrangement (second legend) achieves 1.36× speedup on average. The scheme gives more benefit to the small models like BERT-TINY (2.05× speedup) as they have a higher chance of underutilization. Also, it gets higher speedup, 1.46× for BERT-MG3 which has the largest irregularity in matrix operations. Next, Vec-unit Reconstruction (last legend) brings 1.17× additional speedup on average. Note that for BERT, whether Vec-unit has an SCS structure (third legend) or an optimal structure (last legend) does not make a difference in performance. The overall average speedup is 1.59×, with a minimum of 1.22× and a maximum of 2.73×.

Fig. 6.2 shows the FlexRun's performance improvement for GPT2 with three beamwidths, 5, 10, and 40. In the case of GPT2, Gemv-unit Rearrangement brings 1.19× speedup on average. Also, Vec-unit Reconstruction gets 1.1× additional speedup so that the overall average speedup is 1.31×. Unlike BERT, SCS structure degrades the performance in GPT2. These differences arise from the presence of dependencies between the inputs. If there are dependencies between the inputs, the pipeline's depth affects the performance. Therefore, the SCS structure harms the performance of GPT2. Also, when we increase the beamwidth of GPT2, the performance degradation of SCS decreases as the inputs without dependencies increase.



(a) Speedup of FlexRun on GX compared to V100 for BERT.

(b) Speedup of FlexRun on MX compared to V100 for GPT2.

Figure 6.3: Comparison of FlexRun and V100 for BERT and GPT2.

6.2 Comparison of FlexRun and GPU

In Fig. 6.3, we compare the results of FlexRun with V100 for BERT and GPT2. The x-axis is a scale of the NLP models and the y-axis is the speedup normalized to the latency of V100. FlexRun achieves 2.79× and 2.59× average performance improvements over V100 for BERT and GPT2, respectively. Especially, FlexRun shows higher performance for the small models as small models suffer severe underutilization in GPU. Also, FlexRun usually gets higher speedup for BERT because pipelining and Vec-unit Reconstruction give more benefits to the encoder structure. As the encoder structure does not have dependencies between the inputs, the pipeline depths are al-

most hidden.

6.3 Scalability of FlexRun

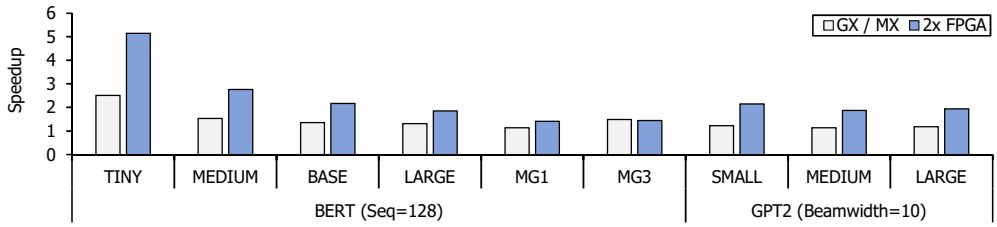


Figure 6.4: Speedup of BERT on GX and FPGA with twice as many resources as GX. & Speedup of GPT2 on MX and FPGA with twice as many resources as MX.

We check the scalability of FlexRun by doubling the compute and memory resources of FPGAs. We assume that the future generation of FPGAs has twice more compute and bandwidth resources than current FPGAs, GX, and MX. Fig. 6.4 is the speedup of FlexRun on the future generation of FPGAs and current FPGAs (GX and MX). The first legend (GX/MX) is the results of GX and MX and the second legend (2× FPGA) is the results of future FPGAs. The speedup of FlexRun on GX and MX (for GPT2) is normalized to the same baseline of Fig. 6.1. For FlexRun on 2× FPGA, we assume a new baseline with twice the #TILE and twice faster memory than the baseline of Fig. 6.1.

The results show that FlexRun achieves scalable performance improvements as FPGA resources increase. In the case of BERT-LARGE, FlexRun attains 1.85× speedup on 2× FPGA while achieving 1.32× speedup on GX. On average, with twice the resources, FlexRun gets 2.2× and 1.99× speedup for BERT and GPT2, respectively. This is 1.46× and 1.69× additional speedup for FlexRun on GX and MX, respectively. The FlexRun secures scalability thanks to Gemv-unit Rearrangement. When the resources increase, the chances of underutilization in gemv compute unit grows due to fragmentation. So for the future FPGAs, it is important to find the proper di-

mension of the gemv compute units, which is done by Gemv-unit Rearrangement in FlexRun.

6.4 Effectiveness of FlexRun

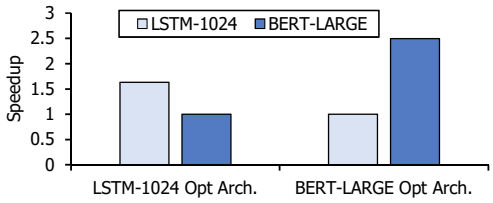


Figure 6.5: Effectiveness of FlexRun.

Lastly, we show the effectiveness of FlexRun in Fig. 6.5. We run BERT-LARGE and LSTM-1024 on architectures optimized for each model using FlexRun. In the x-axis, LSTM-1024 Opt Arch is the architecture optimized for LSTM-1024 and BERT-LARGE Opt Arch is the architecture for BERT-LARGE. The performance of the models is normalized to the slower one. In Fig. 6.5, the performance is severely compromised when the model is executed on the atchitecture optimized for other model. In the BERT-LARGE case, the performance improves 2.83× when running on BERT-LARGE Opt Arch than on LSTM-1024 Opt Arch. For LSTM-1024, the performace improves by 1.63× on its optimized architecture.

Chapter 7

Realted Work

There are works that aim to accelerate NLP models. Each study exploits different methods to achieve their purposes.

First, there are studies using the quantization method to accelerate NLP models and reduce models' sizes [33], [34], [35]. [33] and [35] suggest new quantization methods, expressing parameters of BERT with 3 bits. Also, [34] presents BERT's parameters with eight bits, targeting INT8.

Similar to quantization, many studies attempt to apply pruning to NLP models. [13] uses the weight pruning to reduce the size of LSTM and designs architecture for sparse LSTM. Also, [31] proposes block-circulant matrices for weight matrices to resolve irregularities in the neural network in addition to pruning. For attention-based NLP models like BERT, [21] proposes a structured pruning while [21] uses structured dropout.

[16], [18], [19], and [17] utilize model partitioning for accerlation. [17] defines parallelizable dimensions in DNNs and finds the best parallelization strategies for the target model. [18] applies holistic model partitioning to all operations across attention-based NLP models. [16] exploits model partitioning to accelerate large RNN models by enabling multi-FPGA executions.

Also, some works design accelerators for the NLP models. [12] targets attention

operations in NLP models and makes attention-specialized units. [11] exploits PIM technologies to minimize the memory overhead of the NLP models. Meanwhile, [10] and [22] exploit FPGAs as their HW platforms to accelerate NLP models. However, none of those works can address all three challenges that NLP models possess.

Lastly, [30], [36], and [37] take modular approach for accelerating DNNs. However, these works focus on Convolutional Neural Networks, rather than NLP models. [30] suggests a modular accelerator generator for CNNs. [36] and [37] use FPGAs to build accelerators through their design space exploration tool in the cloud and edge-computing environments.

Chapter 8

Conclusion

In this paper, we propose FlexRun, an FPGA-based modular architecture approach to accelerate NLP models. When receiving input models, FlexRun reconfigures the architecture adaptively to the models. In evaluation, we get 2.69× and 1.44× performance improvement compared to V100 and Brainwave-like FPGA baseline, respectively.

Bibliography

[1] "Nvidia Tesla V100 GPU Architecture, The World's Most Advanced Data Center GPU." 2017. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[2] "Deeplearningexamples," https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow, 2021.

[3] "Nsight systems release notes," https://docs.nvidia.com/nsight-systems/ReleaseNotes/index.html, 2021.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[5] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[6] A. Boutros, E. Nurvitadhi, R. Ma, S. Gribok, Z. Zhao, J. C. Hoe, V. Betz, and M. Langhammer, "Beyond peak performance: Comparing the real performance of ai-optimized fpgas and gpus," in *2020 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2020, pp. 10–19.

[7] Chris Leary and Todd Wang, "XLA: TensorFlow, compiled." 2017. [Online]. Available: https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html

[8] L. B. M. Deo and J. Schulz, "Intel® stratix® 10 mx devices with samsung* hbm2 solve the memory bandwidth challenge," 2019.

[9] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *arXiv:1810.04805*, 2018.

[10] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi *et al.*, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *ISCA*, 2018.

[11] H. Guo, L. Peng, J. Zhang, Q. Chen, and T. D. LeCompte, "Att: A fault-tolerant reram accelerator for attention-based neural networks," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 2020, pp. 213–221.

[12] T. J. Ham, S. J. Jung, S. Kim, Y. H. Oh, Y. Park, Y. Song, J.-H. Park, S. Lee, K. Park, J. W. Lee *et al.*, "Aˆ 3: Accelerating attention mechanisms in neural networks with approximation," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 328–341.

[13] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang *et al.*, "Ese: Efficient speech recognition engine with sparse lstm on fpga," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 75–84.

[14] Y. He, T. N. Sainath, R. Prabhavalkar, I. McGraw, R. Alvarez, D. Zhao, D. Rybach, A. Kannan, Y. Wu, R. Pang *et al.*, "Streaming end-to-end speech recognition for mobile devices," in *ICASSP 2019-2019 IEEE International Conference*

on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2019, pp. 6381–6385.

[15] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[16] H. Jang, J. Kim, J.-E. Jo, J. Lee, and J. Kim, "Mnnfast: A fast and scalable system architecture for memory-augmented neural networks," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 250–263.

[17] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," *arXiv preprint arXiv:1807.05358*, 2018.

[18] J. Kim, S. Hur, E. Lee, S. Lee, and J. Kim, "Nlp-fast: A fast, scalable, and flexible system to accelerate large-scale heterogeneous nlp models," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2021, pp. 75–89.

[19] D. Kwon, S. Hur, H. Jang, E. Nurvitadhi, and J. Kim, "Scalable multi-fpga acceleration for large rnns with full parallelism levels," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[20] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, "Nvidia tensor core programmability, performance & precision," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 522–531.

[21] J. McCarley, R. Chakravarti, and A. Sil, "Structured pruning of a bert-based question answering model," *arXiv preprint arXiv:1910.06360*, 2019.

[22] E. Nurvitadhi, D. Kwon, A. Jafari, A. Boutros, J. Sim, P. Tomson, H. Sumbul, G. Chen, P. Knag, R. Kumar *et al.*, "Why Compete When You Can Work Together: FPGA-ASIC Integration for Persistent RNNs," in *FCCM*, 2019.

[23] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[24] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," *arXiv preprint arXiv:1606.05250*, 2016.

[25] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou *et al.*, "Mlperf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 446–459.

[26] D. E. Rumelhart and J. L. McClelland, *Learning Internal Representations by Error Propagation*, 1987, pp. 318–362.

[27] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter," *arXiv preprint arXiv:1910.01108*, 2019.

[28] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[29] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention Is All You Need," in *NIPS*, 2017.

[30] R. Venkatesan, Y. S. Shao, M. Wang, J. Clemons, S. Dai, M. Fojtik, B. Keller, A. Klinefelter, N. Pinckney, P. Raina *et al.*, "Magnet: A modular accelerator generator for neural networks," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–8.

[31] S. Wang, Z. Li, C. Ding, B. Yuan, Q. Qiu, Y. Wang, and Y. Liang, "C-lstm: Enabling efficient lstm using structured compression techniques on fp-

gas," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 11–20.

[32] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[33] A. H. Zadeh, I. Edo, O. M. Awad, and A. Moshovos, "Gobo: Quantizing attention-based nlp models for low latency and energy efficient inference," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 811–824.

[34] O. Zafrir, G. Boudoukh, P. Izsak, and M. Wasserblat, "Q8bert: Quantized 8bit bert," *arXiv preprint arXiv:1910.06188*, 2019.

[35] W. Zhang, L. Hou, Y. Yin, L. Shang, X. Chen, X. Jiang, and Q. Liu, "Ternarybert: Distillation-aware ultra-low bit bert," *arXiv preprint arXiv:2009.12812*, 2020.

[36] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "Dnnbuilder: an automated tool for building high-performance dnn hardware accelerators for fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[37] X. Zhang, H. Ye, J. Wang, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "Dnnexplorer: a framework for modeling and exploring a novel paradigm of fpga-based dnn accelerator," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.

초 록

최근 딥러닝 기반의 자연어 처리 모델이 음성인식, 번역과 같은 자연어 처리 과제에 적극적으로 활용되고 있다. 자연어 처리 과제는 주로 즉각적인 반응을 요구하기 때문에 단일 배치 환경에서 자연어 처리 모델의 빠른 추론을 지원하는 것이 필수적이다. 하지만 자연어 처리 모델이 가진 특성들로 인해 단일 배치에서 자연어 처리를 가속하는 것은 힘들다. 해당 특성들은 다음과 같다; (1) 넓은 범위의 디멘션과 불균형한 매트릭스 연산, (2) 벡터 연산의 오버헤드, 그리고 (3) 벡터연산의 다양성. 본 학위논문에서는 FlexRun을 제안하여 세 가지 특성들을 해결하고 단일 배치 환경에서 자연어 처리의 추론을 가속한다. FlexRun은 FPGA의 높은 reconfigurability를 활용하여 주어진 타깃 모델에 맞게 아키텍처를 디자인한다. FlexRun에는 세 가지 기술이 있다. 첫 번째는 FPGA를 기반으로 하며 재구성 가능한 요소들로 이루어진 베이스 아키텍처 템플릿이다. 두 번째는 디자인 스페이스를 정의하고 디자인 스페이스에서 타깃 모델에 따라 최적의 디자인 포인트를 찾는 알고리즘이다. 마지막으로는 최적의 디자인을 찾는 것에서부터 아키텍처를 구현하는 일련의 과정들을 자동화하는 툴이다. 본 논문에서는 FlexRun을 적용하여 GPU 베이스라인과 FPGA 기반의 Brainwave-like 베이스라인과 비교해 유의미한 성능향상을 보여준다.

주요어: 딥러닝, 자연어 처리, FPGA, 모듈러 아키텍처, 가속기 설계/디자인, 하드웨어 아키텍쳐, 디자인 스페이스 탐색

학번: 2019-24165